Today: Dynamic Programming IV (of 4)
- 2 kinds of guessing
- piano/guitar fingering (& DDR, RBVGH, ...)
- trees: longest path, carving, skills

* 5 easy steps to dynamic programming:
  1. define subproblems
  2. guess (part of solution)
  3. relate subprob. solutions
  4. recurse & memoize
  5. solve original problem: = a subproblem

* 2 kinds of guessing:
  A: in 1, guess which other subproblems to use
  (used by every DP except Fibonacci)
  (also sometimes used in 5)
  B: in 1, create more subproblems to guess/
  remember more structure of solution
  (used by knapsack DP)
  - effectively report many solutions to subprob.
  - lets parent subproblem know features of sol.
Piano/guitar fingering:

- Given musical piece to play, say sequence of $n$ (single) notes with right hand
- Fingers $1, 2, \ldots, F=5$ for humans
- Metric $d(f, p, g, q)$ of difficulty going from note $p$ with finger $f$ to note $q$ with finger $g$
  e.g. $1 < f < g$ & $p > q \Rightarrow$ uncomfortable stretch rule: $p \leq q \Rightarrow$ uncomfortable legato (smooth) $\Rightarrow \infty$ if $f=g$
  weak-finger rule: prefer to avoid $g \in \{4, 5\}$ 3$\rightarrow$4 & 4$\rightarrow$3 annoying $\sim$ etc.

First attempt:

1. Subproblem $= \min$ difficulty for suffix notes[i:]
2. Guessing $= \text{finger } f \text{ for first note[i]}
3. Recurrence:
   
   $DP[i] = \min(DP[i+1] + d(\text{note[i]}, f, \text{note[i+1]}, ?))$ for $f$
Correct DP:

1. **subproblem** = min. difficulty for suffix notes[i:] starting with finger f on first note[i]  
   \[ \Rightarrow n \cdot F \] subproblems

2. **guessing** = finger g for next note[i+1]  
   \[ \Rightarrow F \] choices

3. **recurrence**:  
   \[ DP[i, f] = \min(DP[i+1, g] + d(note[i], f, note[i+1], g) \]  
   for g in range(F) \]

   \[ DP[n, f] = \emptyset \]  
   \[ \Rightarrow \Theta(F) \] time/subproblem

4. **topo. order**: for i in reversed(range(n)):
   - for f in 1, 2, ..., F:

   - total time: \( \Theta(nF^2) \)

5. **orig. prob.** = \[ \min(DP[\emptyset, f] \] for f in 1, ..., F)  
   (guessing very first finger)

**DAG:**

```
notes
```

```
fingers
```

```
difficulty
```
Guitar: up to $S$ ways to play same note!
- redefine "finger" = finger playing note + string playing note

$\Rightarrow F \rightarrow F \cdot S$

**Generalization:** multiple notes at once (e.g. chords)
- input: $\text{notes}[i] = \text{list of } \leq F \text{ notes}$ (can't play $>1$ note with a finger)
- state we need to know about "past" now assignment of fingers to notes/null

$\Rightarrow (F+1)^F$ such mappings

1. $n \cdot (F+1)^F$ subproblems
2. $(F+1)^F$ choices (how $\text{notes}[i]$ is played)
3. $n \cdot (F+1)^{2F}$ total time

- works for 2 hands ($F=10$)
- just need to define appropriate $d$

**Video games:**
- Guitar Hero / Rock Band:
  just like piano/guitar ($F=4, 1$ string)
- Dance Dance Revolution:
  just like multifinger ($F=2$ feet)
We've seen lots of examples on ~ strings. Trees also often amenable to DP.

Longest simple path in a tree (NP-hard in general graphs)
- must be leaf to leaf
- in rooted tree, must go up then down
- transition where? Guess t ∈ V

1. subproblems = longest path from u ∈ V down to a leaf = h(u)
   - this is actually the height of node v
2. guess first edge (u,v) of path
3. recurrence: h(u) = \max(w(u,v) + h(v) for v in u.children)
4. order: post-order traversal
5. solution = \max(\text{sum of 2 max} (w(u,t) + h(t)) for u in t.children) for t in V)

- time: 3 & 5 cost \leq \sum_{t \in V} O(\# \text{children of } t) = O(n)

* for tree problems, good subproblem choice = rooted subtrees
Tree carving: [Gil&Itai 1999]

- goal: divide given rooted binary tree into subtrees of size $\leq B$ ("blocks") to minimize max./avg. # blocks visited by a root-to-leaf path.

- motivation: when fetching 1 node from disk $\rightarrow$ memory or memory $\rightarrow$ cache, usually get whole block of $B$ nodes (containing requested node) for same cost.

~ so how to optimally store a given tree?

First attempt:

1. **Subproblem** = subtree rooted at node $v$
2. **guess**: which $B$ nodes to put in root block (containing $v$)

- problem: $2^{\Theta(B)}$ choices

- so need to choose block node by node...
- after root node, have 2 subtrees contributing to root block
- essentially, have a resource (root block capacity) & need to divvy it up among children... by GUESSING! 

[Diagram of a triangle with nodes and edges labeled B/2, or, or, or, with $\Theta(B)$ independent choices]
Correct DP:

1. **subproblems**: \( C(v, X) = \text{cost of subtree rooted at } v, \text{ given room for } X \text{ "free" nodes} \)
2. **guess**: how to split \( X-1 \) remaining free nodes among \( v \)'s children
3. **recurrence**:
   \[
   C(v, X) = \min_{Y > 0} \left( C(v, \text{left}, Y) \times C(v, \text{right}, X-1-Y) \right)\]
   max, or weighted avg. for \( Y \) in range({\(X\)})

   \[
   C(\text{None}, X) = \emptyset \\
   C(v, 0) = 1 + C(v, B)
   \]
4. **order**: for \( v \) in post-order traversal:
   for \( X = B, B-1, \ldots, 0 \) (need \( B \) before \( 0 \))
5. **goal**: \( C(\text{root}, 0) \)

- **time**: \( nB \) subproblems \( \cdot \leq B \) guesses \( \cdot O(1) \) time/guess = \( O(nB^3) \leq O(n^3) \)

**Extensions:**
- **nonbinary tree**:
  - split high-degree node into binary tree
  - treat added nodes as free (\(X-Y\), no -1)
- faster, approximate DP algorithms:
  - \( O(nB) \) time, additive error \( \leq 1 \)
  - \( O(n) \) time, additive error \( \leq 1+\varepsilon \)

[Alstrup, Bender, Demaine, Farach-Colton, Rauhe, Thorup 2003]
Skill build tree:  (MMOs, StarCraft, tower def,...)
- each node $v$ costs $c(v) \in \{0, 1, \ldots, C\}$, has benefit $b(v) \in \mathbb{R}$, and limit $l(v)$ on # times it can be bought
- can only buy node if bought parent
- upgrade nodes: # purchases of node $\leq$ # purchases of parent
- goal: maximize benefit with cost $\leq C$

1. subproblems: $\text{OPT}(v, X, l) = \max$ benefit on subtree rooted at $v$, with $v$ buyable $l$ times, given budget $X \in \{0, 1, \ldots, C\}$
2. guess: # times $k$ to buy $v$ & budget allocation $Y$ for left child
3. recurrence: $\text{OPT}(v, X, l)$  
   = $\max$ (  
   $\text{OPT}(v, \text{left}, Y, \{l \text{ if } v, \text{left} \text{ is upgrade} \}$, 
   $\{l(v, \text{left}) \text{ else} \}$  
   + $\text{OPT}(v, \text{right}, X-Y-k \cdot c(v), \{l \text{ if upgrade} \}$, 
   $\{l(v, \text{right}) \text{ else} \}$ 
   + $k \cdot b(v)$
   for $k$ in $1, 2, \ldots, l$
   for $Y$ in $0, 1, \ldots, X-k \cdot c(v)$
   )

$\text{OPT}(\text{None}, X, l) = \emptyset$  
if no option

4. order: for $v$ in post-order traversal:
   for $X$ in $0, 1, \ldots, C$:
   for $l$ in $0, 1, \ldots, l(v)$:

5. $\text{OPT}(\text{root}, C, l(\text{root}))$